**알고리즘 설계와 분석 HW4**

컴퓨터공학과 20171666 이예은

1. 실험 결과

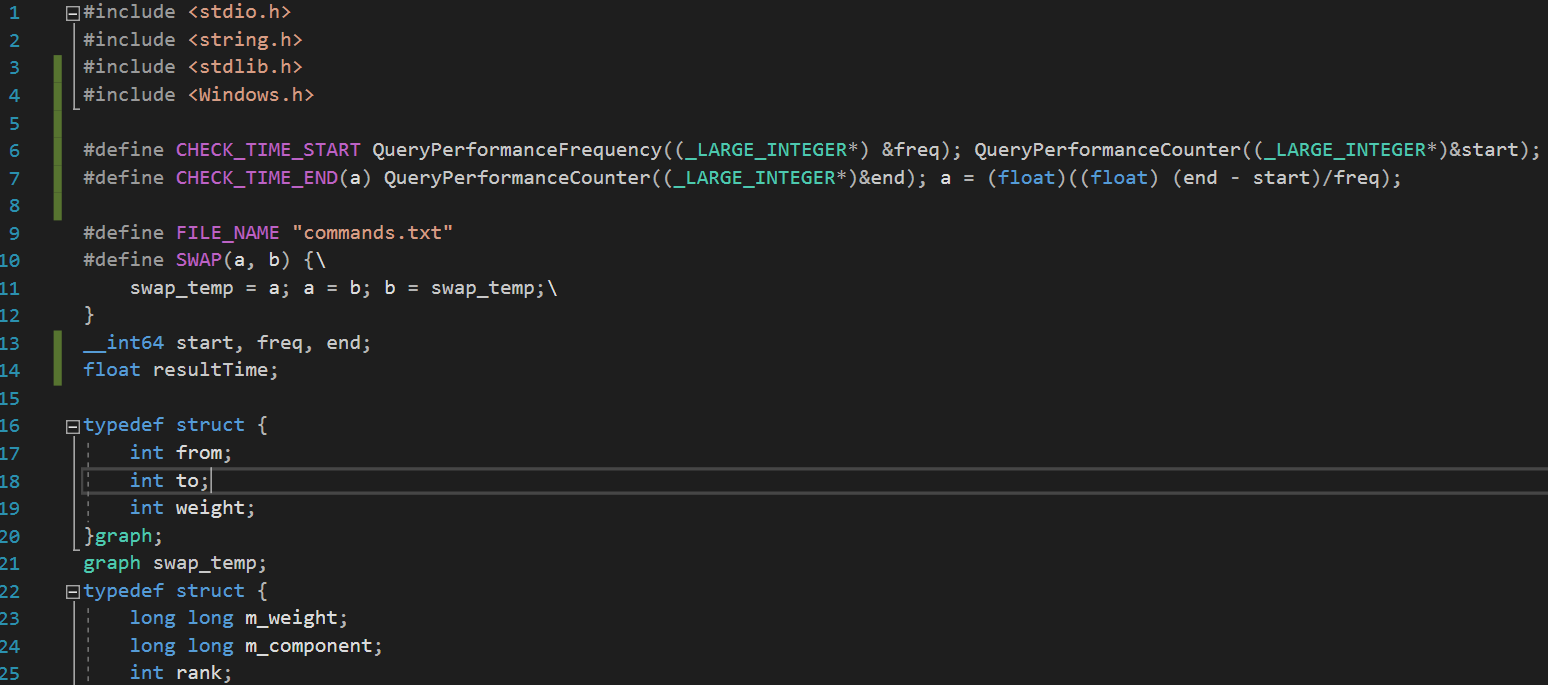
|  |  |  |
| --- | --- | --- |
| 파일 이름 | 작동 여부 | MST weight |
| 1. HW4\_email-Eu-core.txt | YES | 3110161 |
| 2. HW4\_com-dblp.ungraph.txt | YES | 2747895457 |
| 3. HW4\_com-amazon.ungraph.txt | YES | 2729670156 |
| 4. HW4\_com-youtube.ungraph.txt | YES | 14578691475 |
| 5. HW4\_wiki-topcats.txt | YES | 5351181035 |
| 6. HW4\_com-lj.ungraph.txt | YES | 28308045762 |

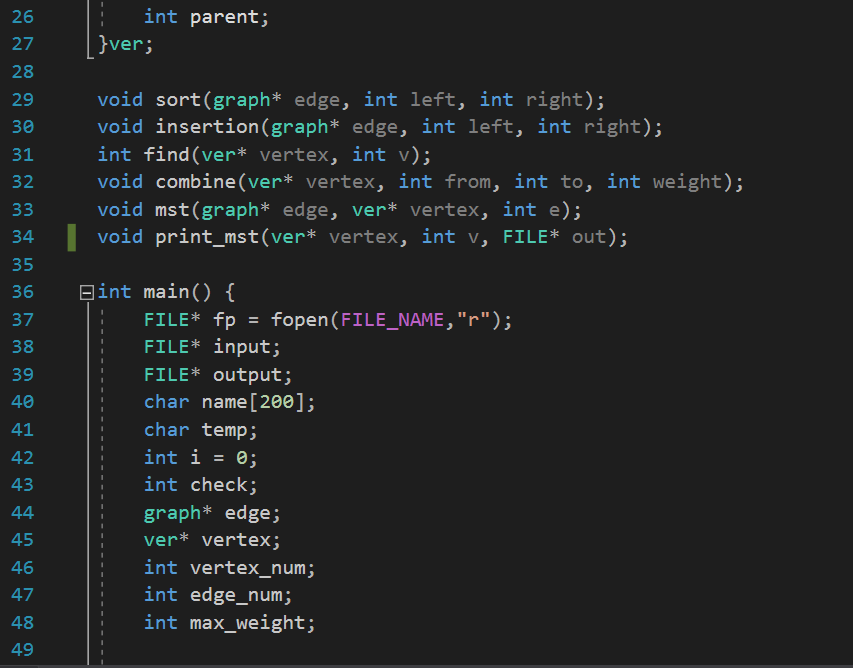
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 수행시간(초) | 1 | 2 | 3 | 4 | 5 | 평균 |
| 1번 파일 | 0.003584  (0.00201) | 0.002997  (0.001989) | 0.0045  (0.003077) | 0.004418  (0.002932) | 0.003916  (0.002519) | 0.00388  (0.002505) |
| 2번 파일 | 0.174681  (0.104992) | 0.177855  (0.107802) | 0.186052  (0.106247) | 0.17803  (0.105659) | 0.190659  (0.111561) | 0.18146  (0.10725) |
| 3번 파일 | 0.178973  (0.101401) | 0.172604  (0.096279) | 0.175580  (0.095174) | 0.172391  (0.095614) | 0.191257  (0.112198) | 0.178161  (0.100133) |
| 4번 파일 | 0.652299  (0.428694) | 0.588130  (0.381292) | 0.596331  (0.381399) | 0.599078  (0.394153) | 0.602588  (0.390385) | 0.607685  (0.395185) |
| 5번 파일 | 12.226767  (9.655131) | 12.316259  (9.676908) | 14.006222  (10.855737) |  |  | 12.84975  (10.06259) |
| 6번 파일 | 16.356905  (12.533333) | 16.369114  (12.555743) | 16.690763  (12.650923) |  |  | 16.470226  (12.58) |

(5,6번은 시간이 너무 오래 걸려 3번만 측정하였다.)

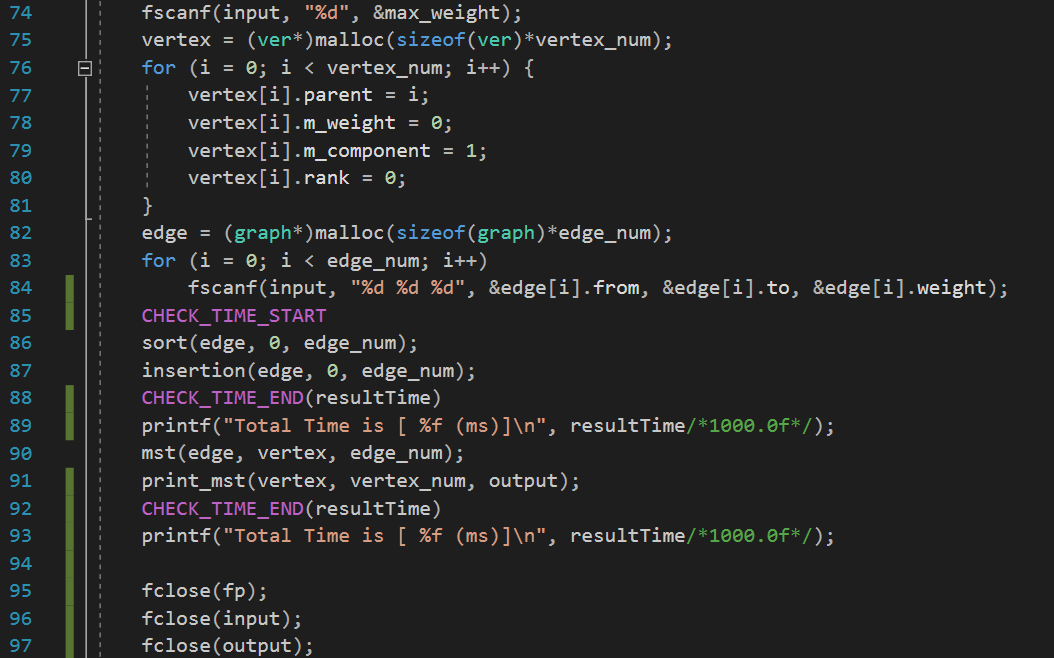
1. 구현 방법

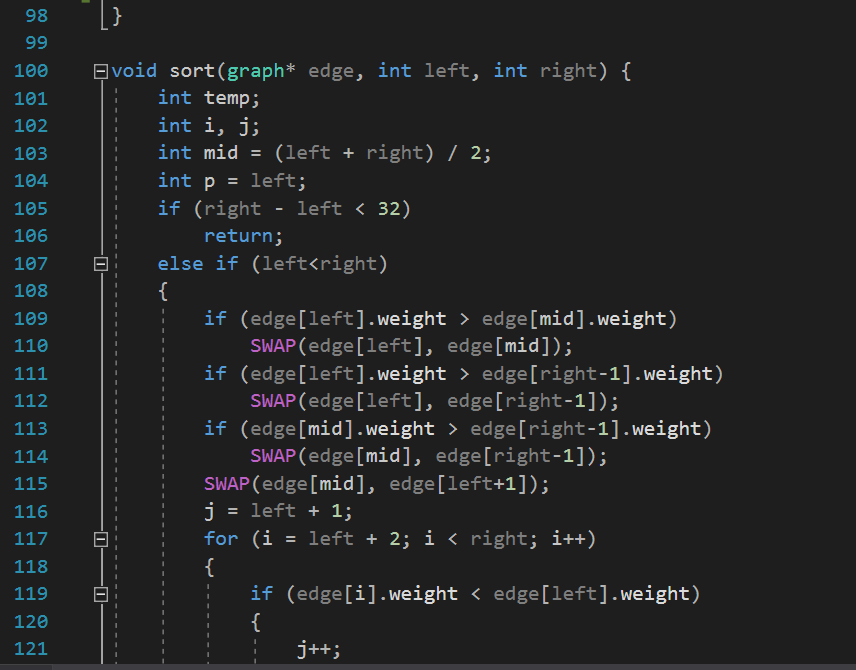
1) 코드

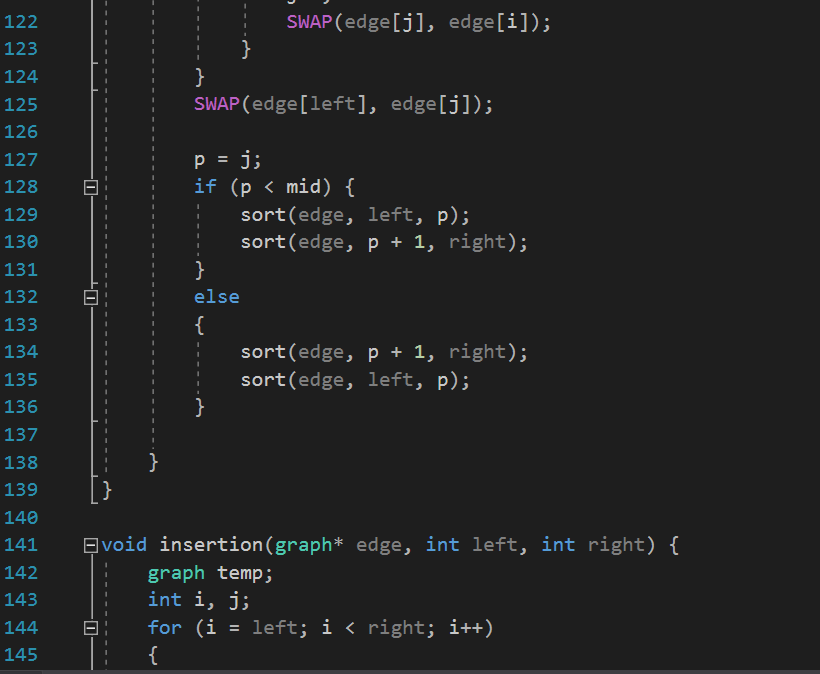


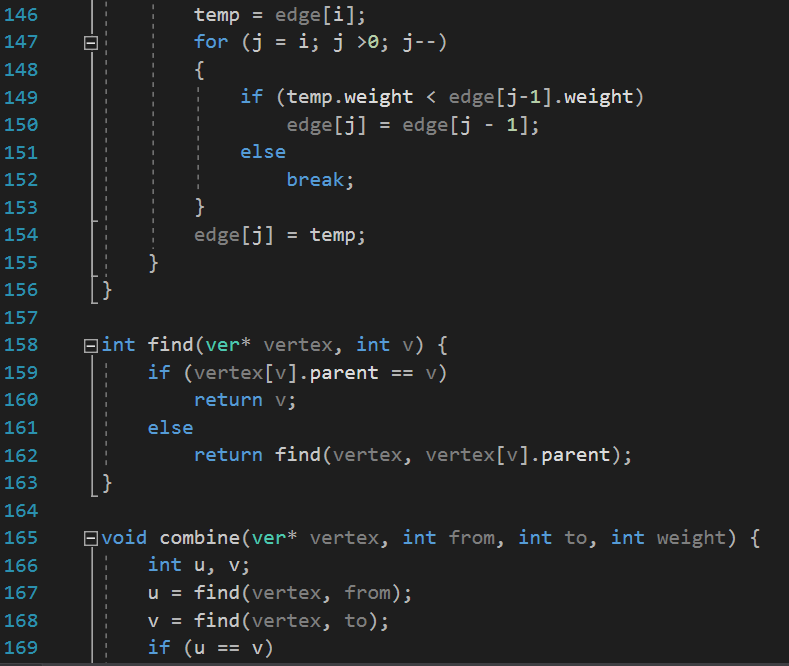


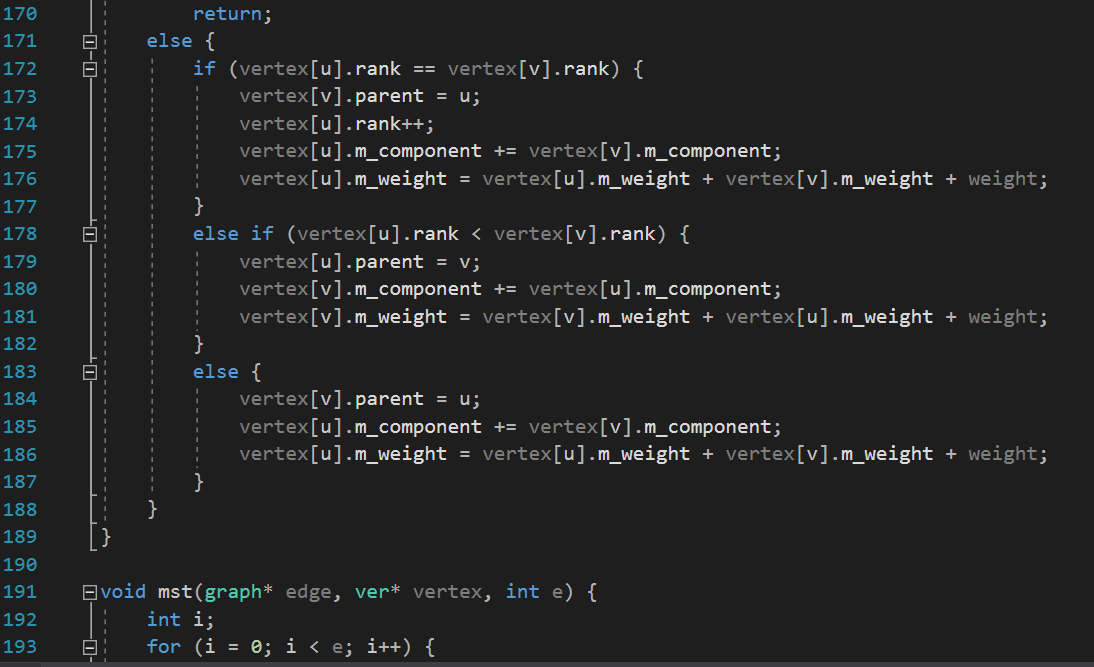


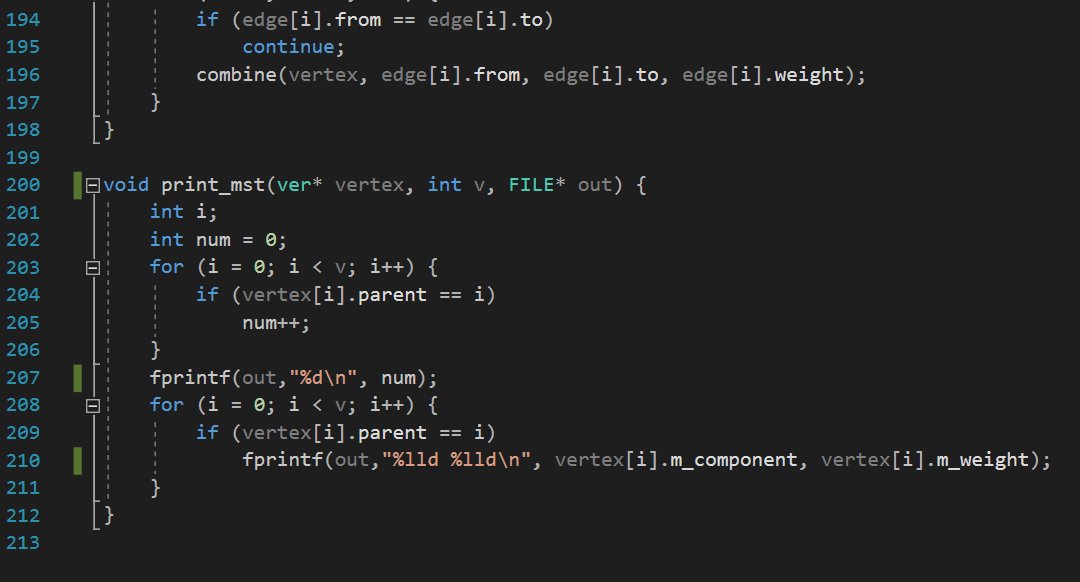












2) 함수

(1) main() : 메인 함수에서는 mst를 실행하기 위해 데이터를 읽는다. Mst를 효과적으로 실행시키기 위해, disjoint set을 실행하기 위해 vertex와 edge의 정보를 저장하는 edge와 vertex 배열은 구조체의 배열로 선언한다. 이 때, edge 구조체 안에는 엣지가 연결해주는 정점 두개의 번호와 엣지의 weight가 들어간다. Vertex 구조체 안에는 disjoint set을 위해 rank와 parent의 번호를 저장한다. Rank는 0으로 초기화하고, parent는 자기 자신의 위치 번호를 저장하여 parent가 자기 자신을 가리키도록 한다. 또한 mst를 만들었을 시 그 때 연결된 component 개수와 weight도 알아야 하기 때문에 vertex 구조체 안에 이를 저장해 줄 변수를 하나씩 생성하였다. 이를 이용해 구조체의 배열을 선언하면 linked list를 사용하지 않더라도 훨씬 쉽게 graph를 표현할 수 있다.

이제 commands.txt 파일을 우선 읽어 input과 output파일의 이름을 받고, 이에 따라 파일을 열어준다. 이 후 데이터 형식에 맞춰 edge 배열에 하나씩 넣어준다. 파일을 전부 입력 받았다면, vertex 배열을 초기화 해준다. 이후 sort() 함수와 insertion() 함수를 불러 edge를 weight에 따라 오름차순으로 정렬해준다. 다음에는 mst()를 호출하여 mst 를 생성하고 print\_mst()를 불러 구한 mst의 component 개수와 weight를 출력해준다.

(2) sort() : sort 함수는 이전 알고리즘 프로그래밍 과제 1번에서 구현한 quick\_sort\_pistro 기법을 적용하였다. 기본적으로 quick sort 를 사용하되, median 기법과 tail recursion 기법을 모두 적용하였고, 크기가 32 이하일 때에는 더 이상 quick sort를 돌지 않고 return 하여 이후 main에서 insertion sort로 넘어가도록 구현하였다. Quick sort 방법으로 시간 복잡도는 이론상 O(eloge)이다. E는 edge의 개수이다.

(3) insertion() : insertion sort 하는 함수로 edge의 weight에 따라 quick sort 에서 정렬하지 않은 부분을 정렬해준다. 대부분 정렬된 배열이기 때문에 시간 복잡도는 거의 O(e)에 가깝다.

(4) find() : disjoint set을 사용하기 위한 함수로 vertex 배열에서 parent를 찾고 싶은 원소의 위치를 받아 parent 의 위치를 반환한다. 만약 vertex 배열 원소의 parent가 자기 자신이라면 그 원소는 root 원소이므로 이를 반환하고, 아닐 경우 자신의 위에 parent가 존재 한다는 것이므로 그것의 위치를 넣어 다시 find()를 불러준다. 이 때, 따로 path compression 기법을 적용하지는 않았다. Parent를 찾을 때 까지 거슬러 올라가므로 O(logv) 만큼의 시간이 걸린다.

(5) combine() : disjoint set의 union함수의 역할을 한다. 두 개의 vertex 번호를 받아 각각의 parent를 find() 함수를 불러 u와 v에 그 parent 번호를 넣어준다. 만약 u와 v가 같다면 둘의 parent가 같다는 것이므로 합치지 않고 바로 return 하여 함수를 끝낸다. 다르다면 세가지 경우로 나뉘는데, u의 rank가 v의 rank와 같은 경우, u의 rank가 v보다 작은 경우, u의 rank가 v보다 큰 경우이다.

같은 경우에는 둘 중 어느 것이든 선택하여 합치면 되지만 이 함수에서는 v가 u로 들어

가도록 구현하였다. 즉, u가 root가 되고 v가 u를 parent로 하도록 코드를 작성했다. Rank가 하나 늘었으므로 u의 rank를 1 더해주고 각각의 set이 합쳐진 것이므로 각각의 component의 개수를 더해 u에 component 개수로 넣어준다. 또한 각각의 weight를 더하

고 u와 v를 연결하는 edge의 weight를 받아 이것도 더해 u의 weight에 넣어준다. 이는 이전까지 u와 v에 연결되었던 edge의 weight들과 새로 연결하려는 edge의 weight를 모두 더해야 전체 weight를 계산할 수 있기 때문이다.

U의 rank가 v의 rank 보다 작은 경우에는 이제 v가 root가 되고 u의 parent는 v가 되도록 설정한다. 이 때에는 rank를 따로 갱신할 필요가 없으므로 하지 않고, v의 component와 weight만 갱신해준다.

U의 rank가 v의 rank 보다 큰 경우에는 u가 root 가 되고 v의 parent 는 u가 되도록 설정한다. 또한 rank를 갱신할 필요가 없고, u의 component와 weight만 새롭게 넣어준다. 각각의 parent를 찾기 위해 find()를 부르므로 시간 복잡도는 O(logv)이다.

(6) mst() : mst를 만드는 함수로 전체 edge를 방문한다. 이는 언제 mst가 만들어지는지 확인하기 번거롭기 때문이고, 만약 서로 unconnected한 각각의 mst가 만들어 질 수도 있기 때문이다. 이미 main에서 정렬 된 edge 배열을 받아 weight가 작은 것부터 combine() 함수를 불러 union을 하거나 하지 않는다. 만약 서로 연결하려는 vertex가 같은 것이라면 따로 combine을 부르지 않고 바로 continue로 다음 엣지를 검사하게 한다. 이는 vertex 가 둘이 같다면 당연히 parent 도 같아 굳이 combine을 불러 union 연산을 하지 않아도 되기 때문이다. Edge 하나당 combine을 부르므로 최악의 경우 combine을 e번 반복한다. 그러므로 시간 복잡도는 O(elogv)가 된다.

(7) print\_mst() : 만들어진 mst의 정보를 출력해주는 함수이다. 모든 vertex를 돌면서 만약 vertex의 parent가 자기 자신이라면 그 vertex는 root이므로 이 vertex 안에 저장된 components 개수와 weight를 txt 파일에 출력해준다.

1. 결과 분석

|  |  |  |  |
| --- | --- | --- | --- |
| 파일 이름 | Vertex 개수 | Edge 개수 | 걸린 시간 |
| 1. HW4\_email-Eu-core.txt | 1,005 | 25,571 | 0.00388(0.002505) |
| 2. HW4\_com-dblp.ungraph.txt | 317,080 | 1,049,866 | 0.18146(0.10725) |
| 3. HW4\_com-amazon.ungraph.txt | 334,863 | 925,872 | 0.178161(0.100133) |
| 4. HW4\_com-youtube.ungraph.txt | 1,134,890 | 2,987,624 | 0.607685(0.395185) |
| 5. HW4\_wiki-topcats.txt | 1,791,489 | 28,511,807 | 12.84975(10.06259) |
| 6. HW4\_com-lj.ungraph.txt | 3,997,962 | 34,681,189 | 16.470226(12.58) |

각각의 edge와 vertex 개수, 걸린 시간을 표로 나타내보면 다음과 같다.

전체 걸리는 시간은 sorting에 mst 만드는 것 까지 합친 시간이다. 그러므로 전체 걸린 시간 = O(eloge + elogv), sorting에 걸린 시간 = O(eloge)이 이론적인 시간이다. 이 때, mst 연산 시 k(scanned)가 아닌 모든 edge를 방문했으므로 최악의 경우인 elogv로 계산한다.

1번 파일을 보면, eloge가 0.0025초 걸렸으므로 elogv는 약 e=25v로 생각하여 계산하면, 약 loge=2logv로 생각할 수 있다. 그러므로 elogv는 (1/2)eloge로 0.0025/2 = 0.00125 정도가 걸린다고 생각할 수 있다. 순수 mst 만드는데 걸리는 시간은 0.00388-0.002505=0.001375로 대략 비슷하다고 볼 수 있다. 이를 보면 mst를 만드는데 이론적인 시간 복잡도를 어느정도 따라가는 것을 볼 수 있다.

그렇다면 2번 파일을 보면, vertex의 개수는 대략 300배, edge의 개수는 대략 40배 늘어난 것을 볼 수 있다. 이를 다시 적용하면, 전체 시간은 e:0.0025 = 40e: x로 하여 비례식을 계산하면 x=0.1로 2번 파일에서 eloge의 연산 시간을 보면 0.107로 둘이 거의 같다. 그러므로 sorting 할 때 이론적인 시간 복잡도가 성립한다. 3번 파일에서는 2번과 vertex의 개수는 거의 차이가 없고 edge의 개수가 약간 줄어드는데, 이로 인해 미세하게 시간이 빨라진 것을 확인할 수 있다.

나머지 파일도 살펴보면 edge의 개수와 vertex의 개수가 매우 커지는데 이 또한 가장 처음 1번 파일의 e와 몇배 차이 나는지를 계산하여 비례식을 세워보면 모두 실험한 값과 비슷한 결과가 나오는 것을 확인할 수 있다. 이처럼 disjoint set으로 O(elogv)의 시간 복잡도가 드는 mst를 성공적으로 구현하였다.